Description

EFFICIENT MULTIPLICATION SEQUENCE FOR LARGE INTEGER OPERANDS WIDER THAN THE MULTIPLIER HARDWARE

TECHNICAL FIELD

The present invention relates to arithmetic processing circuitry, specifically multiplier hardware, and methods of operating the same to carry out multiply or multiply-accumulate operations (and related squaring operations) involving at least one multi-word operand that is wider than the multiplier hardware.

The invention relates in particular to hardware control of the multiplication sequence for executing such multi-word operations in an efficient manner, where the method is characterized by the particular order in which the words of the operands are handled.

BACKGROUND ART

Multiplication hardware necessarily has a finite size, typically defined as having a pair of single-word operand inputs and a two-word result output. In order to also carry out multiply-accumulate operations, the multiplier output is normally connected to an accumulator circuit, which is at least two-words plus one-bit wide. (The supplemental bit can be part of the result or simply be present as CARRY information indicating either an overflow in the case of addition or an underflow in the case of subtraction in the accumulate part of the operation.) The basic operation is thus $R = Z \pm XY$. For simple multiplication, $R = XY$, the accumulator input $Z = 0$. For squaring operations, $X = Y$. The basic operation is usually designed to perform standard integer arithmetic, but multiplication hardware

that performs polynomial arithmetic also exists, especially for use in cryptographic applications.

In cryptography and a number of other applications, there is need to multiply very big integers comprising a large number of words. In order to perform these operations using operands that are much wider than the multiplication hardware, the operands must be sliced into a plurality of single-word segments and fed into the hardware in some specified sequence. These segments are operated upon and the intermediate results are accumulated such that the final product is computed as a sum of cross-products of various weights. The word-wide operand segments, as well as the partial results, are stored in a memory that is addressed by the multiplier hardware's operations sequencer.

A typical sequence keeps a first operand's segment constant while the other operand's segments are scanned one word at a time into the multiplier; then the first operand increments to the next word-wide segment and the scan of the second operand is repeated. If $X = \Sigma_i x_i w^i$, $Y = \Sigma_j y_j w^j$, and $Z = \Sigma_k z_k w^k$, with $w = 2^n$, then $R = \Sigma_k r_k w^k = Z \pm XY = \Sigma_k z_k w^k \pm \Sigma_i \Sigma_j (x_i y_j) w^{i+j}$, where $i+j = k$, and where n is the word size in bits. Thus, in a typical operations sequence, the words y_j are cycled over all j for a fixed word x_i, then i is incremented by one and the cycle of words y_j is repeated for the new x_i.

While the above described sequence is straightforward, easy to program, and obtains the correct result, each step or cycle requires an average of three accesses of the random-access memory. In particular, each step requires that y_j and z_k be read from memory, and a partial result r_k be written back to memory.

An object of the invention is to provide a more efficient multi-word multiplication sequence for large integer operations, that requires an average of only one memory access per multiplication.

DISCLOSURE OF INVENTION

The object is achieved by a method that processes the multiply sequence in groups of two adjacent result word-weights (referred to as columns). Within a group of column pairs, the sequence proceeds by alternating columns with steadily increasing (or steadily decreasing) operand segment weights (referred to as a zigzag pattern), so that one of the segments from a preceding multiply cycle is also used in the present multiply cycle, and, except possibly for the first multiply cycle in a given group, only one of the operand segments needs to be read from memory in any multiply cycle for that group. Additions of partial products of the same results weight are performed in an accumulate operation that is pipelined with the multiply operation. Two-word segments of a separate accumulate parameter may also be added at the beginning or end of a sequence of accumulate operations of the corresponding group.

A multiply-accumulate (MAC) unit performs the multiply and accumulate cycles as directed by a firmware-preprogrammed operations sequencer in the multiplication hardware, with reads of operand segments from random-access memory (RAM) being transferred to the MAC unit through registers internal to the multiplication hardware. Likewise, writes of intermediate and final results are conducted back into the internal register for accumulate parameter segment and ultimately back into the RAM.

A further improvement adds internal cache registers for holding frequently used parameter segments that recur at the beginning or end of each group of multiply cycles.

The invention has an advantage over prior multiply sequences in largely matching the multiplier's requirements for parameter segments to the memory access bandwidth of one read and one write per multiply cycle, so that an overall multiplication of large multi-word integers is performed efficiently in a minimum of cycles. The multiplication operation can also be spread over several cycles (using pipelining), in which case it is still considered that, on average, one multiplication is performed per cycle, since one multiplication result is available per cycle and since one new multiplication operation can be started per cycle.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a schematic plan view of the main architecture of a processing system that includes a multiplier engine of the present invention.

Fig. 2 is an interface diagram showing the registers and memory interface for a typical multiplier engine in accord with the present invention.

Fig. 3 is a more detailed schematic plan view of a MAC unit in the multiplier engine of Fig. 1.

Fig. 4 is a chart of plural word-by-word multiplications and additions laid out by operand and result weights for use in illustrating the operation sequence according the present invention.

Figs. 5a, 5b and 5c are a table showing an operation sequence for one multiplication embodiment of the present invention, using an additional set of 5 cache registers storing frequently used operand segments.

Fig. 6 is a chart of plural word-by-word multiplications and additions laid out as in Fig. 4 by operand and result weights for an exemplary "rectangular" multiply-accumulate operation with different size operands.

Figs. 7a, 7b and 7c are a table showing another example of an operation sequence corresponding to the examples of Fig. 6, in this case with a set of 7 cache registers in the hardware.

BEST MODE OF CARRYING OUT THE INVENTION

With reference to Fig. 1, the main architecture of a processing system is seen to include a main core processor 11 and a multiplier engine 13 sharing a random-access memory or RAM 15 and a multiplier control registers cache 17. The multiplier engine 13 includes a multiply-accumulate (MAC) unit 21, an operations sequencer 23 connected to send command signals to the MAC 21, control registers 25 and internal data registers 27.

Memory management/memory protection units (MMU/MPU) 14 and 19 interface the RAM 15 and cache 17 with the processor 11 and multiplier engine 13. In our preferred implementation, there is an MMU/MPU for the processor core 11 concerning both RAM and peripheral accesses (in order to control/limit the accesses to certain zones/peripherals). Here, the multiplier engine 13 is considered a peripheral. Because the multiplier engine 13 has direct access to RAM 15, it could be a way for a user to overcome access limitations specified in the core-side MMU/MPU 19. Therefore, we provide another MMU/MPU 14 to control memory accesses by multiplier engine 13. The two MMU/MPU units 14 and 19 should be configured in a consistent way, but there is no link between them and their operation is independent.

The multiplier engine 13 typically does not have any dedicated ROM, but is configured and parameterized by the processor core 11. The control registers 25 are connected to the control registers cache 17 from which it receives commands from the processor core 11. The control registers 25 transmit command parameters and status information to the operations sequencer 23, and also interact with the MAC unit 21, e.g. to select the MAC mode (standard arithmetic or polynomial arithmetic) for those MAC units that may be capable of both types of arithmetic, to select single-word or multi-word MAC operation, and to communicate any carry values from a current or previous MAC operation.

The internal data and address registers 27 are connected to the shared RAM 15 to receive and transmit operand parameters of a MAC operation. The operations sequencer 23 is preferably pre-programmed with firmware according to the invention described herein. The operations sequencer 23 sends commands and addresses to the internal registers 27 and thence to the shared RAM 15 to direct the loading of selected word-wide operand segments in a specified order according to the present invention. The architecture is typically constructed so that, when the multiplier engine 13 is running, it has privileged access to a specific part of the shared RAM 15. This permits the core 11 to still access other parts of the RAM 15 during a computation. Alternatively, access to the shared RAM 15 could be entirely dedicated to the multiplier engine 13 during a computation and accessible to the processor core 11 only when the multiplier 13 is not using it.

The MAC unit 21 may be based on a 32-bit word size. In this case, operand lengths are always a multiples of 4 bytes and operands are aligned on 32-bit

word boundaries with leading zeros when necessary. This choice facilitates address computations for the shared RAM 15 since the processor 11 typically works on byte addresses. The MAC unit 21 would have a pair of 32-bit (word-wide) operand inputs X and Y for the multiplier array, a two-word wide accumulator input Z, a two-word wide multiplier results output that forms a second accumulator input, and a two-word wide accumulator output R.

Although the MAC unit 21 only operates on single-word operand inputs X and Y, the overall multiplier engine 13, including the programmed operations sequencer 23, can be considered as a large (multi-word) integer multiplier. It supports efficient multiply operations such as multiply-accumulate of N-word numbers, square-accumulate of N-word numbers (multiplier input Y = X), multiply or square of N-word numbers without accumulate (accumulator input Z = 0), and multiply-accumulate of an N-word number by a 1-word (or even 1-byte) constant A.

With reference to Fig. 2, the interaction between the multiplier engine 13 with its various registers and caches and with the shared RAM 15 is illustrated by an interface diagram. The RAM 15 stores the parameters X, Y, Z and R at specific address blocks that must be pointed to in order to access them. The words of the operands are always stored least significant bit first in the RAM, beginning with a base address. In order to request a parameter, or a specific operand segment word, the corresponding address register X ADDR, Y ADDR, Z ADDR or R ADDR (some of the internal registers 27 in Fig. 1) must be loaded with the related address.

The addressed words are then loaded to or from the corresponding data register RX, RY, RZ or RR (more of the internal registers 27 in Fig. 1 used by the MAC unit 21).

Registers 25 typically include one or more operations registers for specifying a desired operation to be performed (multiply, multiply-accumulate, multiply-subtract, multiply by one-word constant, squaring, etc.), one or more control registers specifying various options (natural or polynomial mode, full or truncated operation, carry or carry input, etc.) as well as indicating operand lengths, and one or more status registers indicating various conditions (busy/idle, an overflow/underflow/zero result, error conditions, etc.)

With reference to Fig. 3, the MAC unit 21 of Fig. 1 is made up of an integer multiplier array 31 receiving single-word operands or more usually single-word segments of larger multi-word operands via data registers RX and RY loaded from the shared RAM. The multiplier array 31 outputs a two-word result of multiplying the input words to an accumulator 33. The accumulator 33 has a size of three words (=96 bits) plus a number of carry bits sufficient to handle a specified maximum operand size. For example, for 512 word operands the longest column has 512 rows of intermediate products to be added, thereby requiring a 9-bit space for the carry sum. This gives a total accumulator width of 105 bits for this instance. The accumulator 33 also receives a two-word input parameter from data register RZ, and outputs a result to a results data register RR. Two-word feedback may be provided from the accumulator output or data register RR back to the accumulate operand register RZ to permit the accumulate operand segment to be updated before the final result in register RR is written back to the RAM. A typical size for the multiply-accumulate

(MAC) unit handles 32-bit words, with 32-bit operand inputs from the registers RX and RY to the multiplier array, and with 64-bit operand multiplier array output, and accumulator inputs and output to and from registers RZ and RR.

Letting A represent a single-word operand or a segment of multi-word operand X that is loaded into the multiplier array 31 from the data register RX, and letting B represent a single-word segment of multi-word operand Y that is loaded into the multiplier array 31 from the data register RY, then $A = \Sigma_i a_i 2^i$, $B = \Sigma_j b_j 2^j$, where a_i and b_j are the individual bits of the operand or segment, and where i and j range from 0 to 31.
For multiplication over the integer field Z_p,

$$AB = \Sigma_i \Sigma_j a_i b_j 2^{i+j}.$$

For multiplication over the Galois field $GF(2^n)$, addition of two bits is reduced modulo 2 so that

$$AB = \Sigma_{k=0}^{2n-2} (2^k \cdot \Sigma_{i+j=k} (a_i \cdot b_j \bmod 2))$$

There may also be a carry-in term W which is added. The handling of the carry depends on the options indicated by the registers 25 mentioned above. Note also that the carry-in term W need not have a direct relationship to the outgoing carry term from an immediately prior computation. Finally, Galois field operation has an influence over carry processing in that the adding of the carry-in term W to the least significant bit is also conducted modulo 2.

Full-scaled multiplication of multi-word operands X and Y involves a sequence of the single-word multiplications just described. X and Y are respectively N-word and M-word numbers. The general operation is

$R = [Z] \pm ((X \cdot Y) + W)$. This can be written in sum-of-products form as:

$$R = [\Sigma_{k=0}^{N+M-1} Z_k b^k] \pm (\Sigma_{i=0}^{N-1} (\Sigma_{j=0}^{M-1} (X_i Y_j b^{i+j})) + W).$$

This formula is valid for both Z_p and $GF(2^n)$ operations, and $b = 2^{32}$. The three-word-plus-carries accumulator is able to compute $Acc := Acc \pm (X_i \cdot Y_j)$ or $Acc := Acc \pm (X_i \cdot Y_j \cdot 2^{32})$, as needed. The invention resides in the particular order in which the single-word multiplies occur.

The Operation Sequence

With reference to Fig. 4, the chart shows a layout of word-by-word multiplication operations of first operand segments A0...A7 of multi-word operand X with the second operand segments B0...B7 of multi-word operand Y. For illustration purposes, both operands X and Y are 8-words wide, but this need not be the case. To obtain the result words R0...R15, the various partial products need eventually to be added vertically together with the corresponding accumulate words C0...C15 of operand Z. In order to memory accesses, the loading of the operand segments, their respective multiplying, and the adding to the accumulate segments, are organized in double columns of adjacent result weights. Each group of two columns is processed starting from either the top or the bottom and progressing line by line in a zigzag fashion. Also, adjacent groups of column pairs need not necessarily progress the same direction. Thus, in Fig. 4, the multiplication sequence could go as: A1B0, A0B0, A0B1; A0B3, A0B2, A1B2, A1B1, A2B1, A2B0, A3B0; A5B0, A4B0, A4B1, A3B1, A3B2, A2B2, A2B3, A1B3, A1B4, A0B4, A0B5; A0B7, A0B6, A1B6, A1B5, A2B5, A2B4, A3B4, A3B3, A4B3,

A4B2, A5B2, A5B1, A6B1, A6B0, A7B0; A7B1, A7B2, A6B2, A6B3, A5B3, A5B4, A4B4, A4B5, A3B5, A3B6, A2B6, A2B7, A1B7; A3B7, A4B7, A4B6, A5B6, A5B5, A6B5, A6B4, A7B4, A7B3; A7B5, A7B6, A6B6, A6B7, A5B7; A7B7. (Semicolons separate the different groups of column pairs in the sequence list.) In this particular example, successive groups progress in opposite directions (top to bottom, then bottom to top, then top to bottom again, etc.) Note that only one operand needs to be read before the array can perform the next multiply, since one operand is already in place from the previous multiply.

If successive groups of column pairs always progress in the same direction, then a number of cache registers can be provided to store certain frequently used operand segments needed to avoid two reads before the first several multiplies at the beginning of each group. In this case, five caches will store A0, A1, B0, B1 and B2 during the ascending half of the multiplication sequence (the right half of Fig. 4, when each successive group becomes longer). The caches then will store A6, A7, B5, B6 and B7 during the descending half of the sequence (the left half of Fig. 4, when each successive group becomes shorter).

Figs. 5a, 5b and 5c illustrates the benefit of having such caches, by showing a multiply sequence for this embodiment of the invention. The first column represents RAM read accesses, the second column represents multiplier array operations, and the five right-most columns show the contents of the five caches during each multiply cycle. For an 8 word by 8 word multiply with 5 caches, there are 64 multiply cycles, 7 cache load-only cycles, 5 other memory read cycles, plus 3 cycles at the end for trailing adds for at total of only 79 cycles. The advantage over prior sequences is

even more apparent when multiplying larger integers common in cryptographic applications (e.g. 1024 words by 1024 bits).

The word size of each operand can be either even or odd, and need not be the same for both operands. The latter case is often referred to as a "rectangular multiply" operation. An example of a rectangular multiply and accumulate operation where one of the multiply operands X has an odd number of words (5) and the other multiply operands Y has an even number of words (14), and where the accumulate operand Z and the result R have an odd number of words (19) is shown in Figs. 6, 7a, 7b and 7c by way of example. In this example, there are also 7 internal cache registers for storing frequently used parameter segments. Only a portion of the total sequence is shown, for groups R0-R1, R2-R3, R4-R5, R6-R7, … (R8 through R11 omitted), R12-R13, R14-R15, and R16-R17. Within the groups, the multiply pattern again is arranged so that at most one RAM read access and one RAM write access is required. The first column shows the specific RAM address that is applied and accessed in successive cycles. The writes to RAM are shown with bold borders. The second and third columns show the operations being executed by the multiplier array and accumulator, respectively. The seven right-most columns show the contents of the internal cache registers used by the multiplier array and accumulator.

This example also shows that the multiply sequence need not be strictly zigzag descending from top to bottom or zigzag ascending from bottom to top over an entire group of column pairs. Rather, the beginnings and/or ends of such a zigzag sequence can be displaced and even proceed in an opposite direction from the first part of a group sequence, because the needed parameters

are available in the caches. Thus, in the first group R0-R1, the order can be Y0X0, Y0X1, Y1X0, instead of Y0X0, Y1X0, Y0X1 for a strictly descending sequence, or instead of Y0X1, Y0X0, Y1X0 for a strictly ascending sequence. In the second, third and fourth groups the sequence begins in the middle at Y1X1, Y3X1, and Y5X1 respectively, descends in zigzag fashion to the bottom, then jumps back up to Y2X1, Y4X1, and Y6X1 respectively and finishes in ascending zigzag fashion to the top. This can be done because operand segment X1 is stored in an internal cache, so still only one read is needed despite the jump from Y0X3 to Y2X1, from Y0X4 to Y4X1, and from Y2X4 to Y6X1 respectively in the middle of the sequence. The same goes for any of the groups omitted in Figs. 7a-7c and for group R12-R13. In the group R14-R15, the sequence begins at the top and descends in strictly zigzag fashion until Y12X3, but then jumps to Y10X4 and finishes in ascending zigzag fashion with Y11X4 and Y11X3. This jump and change of direction in the sequence is permitted because the parameter X4 is already available in a cache register. The final multiply group R16-R17 is shown in a strictly descending zigzag pattern, but can proceed in any order because operand segments X3 and X4 are available in cache registers.

Squaring operations and square-accumulate operations proceed in the same manner as any of the preceding examples, except that the X and Y operands are identical. However, because the specific segments of X and Y are not usually the same in any particular multiply cycle, the square operations can be treated just like any other multiply operation in which the operands X and Y differ.